

Binary Translation and Architecture Convergence Issues for IBM System/390

Michael Gschwind, Kemal Ebcioglu, Erik Altman, Sumedh Sathaye

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

ABSTRACT

We describe the design issues in an implementation of the ESA/390 architecture based on binary translation to a very long instruction word (VLIW) processor. During binary translation, complex ESA/390 instructions are decomposed into instruction “primitives” which are then scheduled onto a wide-issue machine. The aim is to achieve high instruction level parallelism due to the increased scheduling and optimization opportunities which can be exploited by binary translation software, combined with the efficiency of long instruction word architectures. A further aim is to study the feasibility of a common execution platform for different instruction set architectures, such as ESA/390, RS/6000, AS/400 and the Java Virtual Machine, so that multiple systems can be built around a common execution platform.

1. INTRODUCTION

We describe the implementation of the IBM System/390 architecture based on dynamic binary translation to dynamically architect instruction sets on a simple, high performance VLIW architecture. This approach is based on DAISY (Dynamically Architected Instruction Set from Yorktown) from IBM Yorktown [1]. Like DAISY, our work represents a dynamic compilation algorithm which can respond to changing program profiles and adapt the code to workload-specific conditions.

In the course of this architecture study, we have addressed a number of System/390 design aspects required to implement a compliant high-performance implementation of this CISC architecture. These problems include self-modifying code, atomicity of complex CISC instructions in the presence of precise interrupts, re-ordering memory operations to achieve high performance while maintaining MP memory consistency, access registers used to increase the range of addressable memory, the pervasive use of register indirect branches and its impact on the predictability of program control flow, a single condition code flag which is set by almost all operations, and the existence of an execute instruction which composes new instruction words and executes them on the fly.

Dealing with such unusual architecture features is quite important in practice, but often ignored by novel architecture researchers. Deal-

ing effectively with these features is of particular importance when dealing with architectural transitions, i.e., if the semantics of an existing program are to be preserved on a new architecture, either in the course of architectural transition (e.g., binary translation from legacy architectures such as VAX or x86 to more modern processors such as Alpha or IA-64) or architectural convergence using a binary translation layer.

We use the DAISY approach to implement multiple different architecture such as IBM System/390, RS/6000 and AS/400 on a common target VLIW processor to achieve architectural convergence. Previous work [1][2][3] has shown the prospect of using a common processor core for implementing multiple ISAs, and thereby reduce the number of cores which have to be implemented to execute code for the different architectures. This paper presents a first evaluation of such a convergence system.

In fact, the VLIW convergence processor core becomes a new open architecture, where other “computer architectures” are software layers on a single, generic engine. By identifying a common set of execution primitives for a simple VLIW architecture, the efficient execution of several different instruction set architectures on a single processor core becomes possible. A convergence platform offers added flexibility, because architectures are implemented through software as opposed to hardware, and cost advantages because fewer processor cores need to be designed, validated, tested and manufactured.

We have performed a detailed trace-based analysis to determine system performance, including cache and TLB effects, translation cost and interpretative overheads. The performance potential is encouraging.

Dynamic optimization and response to changing program profiles is particularly important for wide issue platforms to identify which instructions should be executed speculatively. Dynamic response as inherent in this described approach offers significant advantages over a purely static compilation approach as exemplified by Intel’s IA-64 architecture. From what has been disclosed, IA-64 relies purely on static profiling which makes it impossible to adapt to program usage.

With the dynamic compilation approach which is included in our platform, software can profile code and schedule in response to changing program profiles, as well as adapt optimization and scheduling to the particular generation of the microprocessor at the core of a binary translation-based system. The internal system architecture is unexposed, i.e., much like microprogram code, it will change with different generations of this system and is not accessible to programmers. The re-optimization capability is particularly important for a platform such as ESA/390, for which a large body of legacy code exists.

We first give an overview of the binary translation system in sec-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 1999 ACM 0-89791-88-6/97/05 ..\$5.00

tion 2. An overview of the architectural features of the target architecture to support the convergence of heterogeneous instruction architectures under binary translation is given in section 3. The performance modeling aspects of binary translation architectures are explored in section 4 and our tool chain is described in 4.2. We discuss preliminary performance results in section 5. We compare our approach with related work in section 6 and draw our conclusions in 7.

2. BINARY TRANSLATION APPROACH

In this section, we describe the execution based dynamic compilation algorithm. In what follows, the “*base architecture*” [4][5] refers to the architecture with which we are trying to achieve compatibility, e.g., *PowerPC* or *ESA/390*.

While previous binary translation efforts have concentrated on achieving an acceptable fraction of the performance of the base architecture, a DAISY-style approach is more ambitious. DAISY aims to implement the emulated architecture with performance significantly better than 1 cycle per instruction. Several strategies are combined to achieve this goal:

- Translation of code to a wide issue machine to achieve high instruction level parallelism. By targeting a wide issue machine, execution bandwidth is increased. Since instructions are scheduled statically, no complex issue and dispatch logic is required.
- Ability to optimize code in the translator. Code can be optimized and tuned based on usage patterns. This is especially important for exploiting the performance potential of new processors when workloads have been compiled using instruction selection and scheduling rules for a different model.
- Dynamic code adaptation through the use of instrumentation and runtime feedback. Frequently executed code can be optimized more aggressively to boost performance based on statistics gathered through hardware based instrumentation. Dynamic code adaptation balances the inflexibilities imposed by static schedules.

Binary translation of *ESA/390* code is based on our experience with the translation of *PowerPC* code to VLIW architectures. Translation occurs incrementally, into acyclic tree-like groups with a single entry point, multiple exits and no joins within a group.

The dynamic translation algorithm interprets code when a fragment of *base architecture* code is executed for the first time. As *base architecture* instructions are interpreted, the instructions are also converted to execution primitives (these are very simple RISC-style operations and conditional branches). These execution primitives are then scheduled and packed into VLIW tree regions which are saved in a memory area which is not visible to the *base architecture*. Any untaken branches, i.e., branches off the currently interpreted and translated trace, are translated into calls to the binary translator. Interpretation and translation stops when a stopping condition has been detected. The last VLIW of an instruction group is ended by a branch to the next tree region.

As each VLIW tree region is translated, a number of optimizations are performed to enhance the available instruction parallelism. These include expansion of register-indirect branches into a series of conditional branches to increase scheduling opportunities [7], copy propagation, combining, load/store telescoping, and unification [8]. Speculation is used aggressively within a translation group, although results are committed in-order to the architected processor state to maintain precise exception behavior.

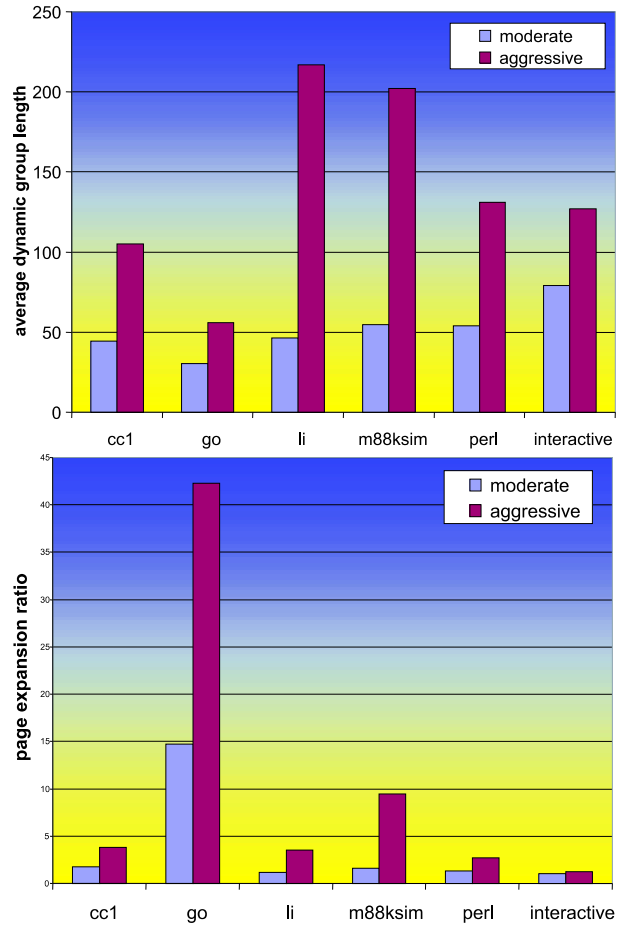


Figure 1: These charts show the impact of group formation aggressiveness on average dynamic path length and code expansion by comparing moderate (mod) and aggressive (agg) group extension policies. More aggressive group extension leads to longer average dynamic group length, but also increases code expansion.

Then, the next code fragment is interpreted and compiled into VLIWs, until a stopping condition is detected, and then next code fragment, and so on. If and when a program transfers control to an entry point of a code fragment for which VLIW code already exists, it branches to the already compiled VLIW code. Recompile is not required in this case.

The compilation of the tree region is not necessarily ever complete. It may have “loose ends” that may call the translator at any time. These calls guard the transition to previously untranslated entry points. When control passes from previously executed code section into new base architecture code the translator starts translation at that point. Thus, dynamic compilation is potentially a never-ending task. The translator is also invoked when a particular code segment is executed frequently. Identification of hot program regions is based on hardware profiling support in the form of a counter array cache. When profiling identifies a hot program region, the translator restarts group formation and translation, this time forming larger groups with more aggressive optimizations.

Increasing the size of groups extends scheduling and optimization possibilities, thereby increasing performance potential. At the same time, it also increases code size by duplicating frequently executed instructions multiple times. This expansion has to be carefully con-

trolled to avoid instruction cache thrashing, or else performance will suffer.

Thus, a delicate balance has to be struck between expanding group size to increase instruction level parallelism and reducing code duplication to achieve good instruction cache performance. Figure 1 details the achievable dynamic group length and resulting code expansion for several SPECint95 benchmark using group formation strategies of varying aggressiveness, as will be described in section 5. Unlike static compilation techniques, the dynamic framework is more flexible in making code duplication decisions and performance trade-offs, since the amount of duplication can be calibrated depending on the workload characteristics.

Having thus described the general framework of our binary translation effort, we turn our attention to issues specific to the translation of CISC architectures, and ESA/390 in particular. Unlike simpler RISC architectures, some CISC instructions can result in long instruction execution times, and represent veritable subprograms. To increase scheduling freedom, we expand CISC instructions into RISC-like operation primitives, as they are interpreted for the first time. Group formation, scheduling, and optimization then occurs on those simpler primitives.

2.1 Resolving branch target addresses

Historically, implementations of the ESA/390 architecture (and its predecessors) have supported only a register-indirect addressing mode for branches.¹ In [7], we have previously described our mechanism for dealing with register indirect branches when translating code from the PowerPC architecture: each register indirect branch is converted into a sequence of conditional branches, testing for previously seen target addresses. This approach allows speculative execution over the multiple targets of indirect branches, and allows code generation to include the (typically few) branch targets when forming groups during translation.

While this approach is appropriate for handling register-indirect branches on the PowerPC architecture, it is inappropriate for ESA/390. This is due to the different usage patterns of register-indirect branch. On PowerPC, register-indirect branches are used for dispatching switch statements, or to call subroutines. As a result, each indirect branch will have its own set of probable register-values.

On the other hand, typical ESA/390 code, particularly that generated by compilers, uses these register-indirect branches by loading a known address (typically the beginning of the current function) into a base register and then using the displacement to specify an offset relative to that known location (beginning of function block) to address branch targets. Thus, the base register for register-indirect branches *virtually always* has the same value within any given function. When using register-indirect branch optimization as described in [7], each register indirect branch is expanded into a sequence consisting of testing the base register, and conditionally branching to the translator if it has changed, before control is passed to the target address (see figure 2).

Since these instructions may be placed in the critical path (just before a basic-block ending branch), this can lead to significant performance penalties. Since the base register value is not expected to change, branching is really independent of the value in the base register. Thus, to achieve optimal performance, tests for the contents of base registers should be eliminated for maximum performance. Unfortunately, *theoretically* the value of the base register could change during the execution, so this assumption cannot be made without determining that it is safe.

¹ A PC-relative branching mode is now supported in the architecture, but many workloads still use the traditional register-indirect branching.

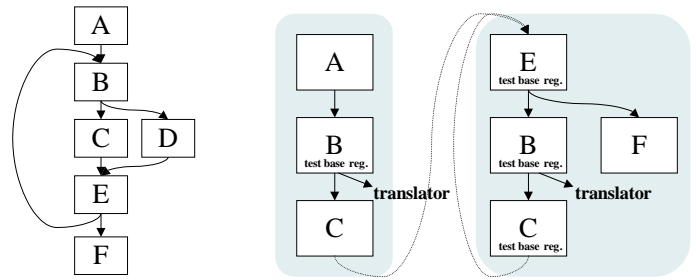


Figure 2: DAISY branch conversion: A flow graph (left) is translated into a sequence of code fragments (right). To resolve register-indirect branches without requiring serialization, these branches are translated into a sequence of tests for known base register values.

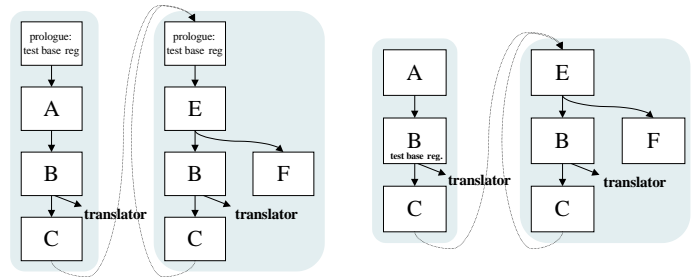


Figure 3: Mimic and DAISY/390 branch conversion: Mimic guards each code fragment with a prolog to check for compile-time assumptions and transfer to the translator if they are violated. In DAISY/390, incremental dataflow information is used to eliminate base register tests.

In [9], May describes the problems associated with translating branches on S/370 and its solution in the context of a problem-state binary translator called Mimic. In Mimic, this is solved by basic flow analysis within a code block. When a code block is entered, a prolog checks the value of the base register just once upon entry into the code block. If the base register has changed, then the translator is invoked to retranslate the code block, otherwise the code block is executed without further checks (see figure 3). This approach allows elimination of a significant number of tests, but still requires tests on every group entry. This penalty can still be quite significant.

DAISY/390 uses an alternative approach: instead of adding a guarding test to each translation unit, we use incremental dataflow analysis between blocks to minimize the need for code which checks compilation assumptions about the contents of base registers.

When a block is translated, dataflow information for the current code block is generated for code optimization techniques performed at the code block level [8]. This includes information such as the constant propagation. This information can then be used when compiling the next code block to perform better optimization. Such optimization includes the elimination of base register compares which would guard conditional branches, or the optimization of other code, such as constant expressions, and so forth.

If such information is used across translation units, special care has to be taken to deal with control flow joins. Since at compilation time, cross-group information is only available for one edge in a control flow join, different information may pass along the new edge. This situation must be detected. When a control flow join at the entry of a translation group occurs, several combinations can occur for dataflow information passing along a newly added edge and the in-

	information carried by new edge		
	X	?	Y
information used to compile group	X	join	test & clone
	?	join	join
	Y	clone	test & clone
			join

Figure 4: Control flow joins with incremental data flow analysis

formation used to compile the group:

As shown in figure 4, each dataflow information item can either have a determined value (denoted as X or Y) or be unknown (denoted as ?). If an edge is added to the entry of an existing translation unit, and the information carried by the new edge is compatible with optimizations based on previous dataflow information, a direct jump from the end of one translation unit to the beginning of the existing translation unit can be used to pass control.

When a conflict between edge-carried dataflow information and previously used dataflow information is detected, a translation group with the same entry address is cloned. The new group can then use different dataflow information to translate the new group. The new group can be compiled using weaker or no dataflow information to reduce the danger of code explosion. Consider an example where constant propagation is used to propagate a value X for some register across a translation group boundary. If at a later time another entry to the same group carries a value Y in the same register, an incorrect result would be produced. To resolve this situation, a new group is generated using either the new value of Y, or without any dataflow information.

If information has been used for optimization of a translation group, and an edge does not carry any dataflow information for an item used in generating the target translation unit, a test may be inserted to query if a particular value matches the value used for code generation of the target group. If the test succeeds, then control can be passed directly to the existing code block. Otherwise, this case is handled as a join with incompatible dataflow information.

The approach outlined here works best for information which rarely carries conflicting information along different edges at a join point. This is the case for base registers in register-indirect branches as used in ESA/390. If frequent, incompatible joins are encountered, code explosion may cause more harm than gain due to excessive translation cloning.

The compatibility checks are easy to perform during binary translation. Each group entry can be associated with a list of assumptions which were used in optimizing the group. Each group exit (tip) also contains a list of relevant control flow information reaching that point. A simple check can then determine compatibility of dataflow at a group exit with information used at the following group entry.

The propagation of base register addresses for register indirect branch resolution is an example of using incremental dataflow analysis to perform constant propagation at runtime. This approach is suitable for all information which is propagated forward along the control flow. Control flow joins with contradictory information are resolved by cloning parts of the control flow graph or retranslation.²

²If code is interpreted several times before attempting translation, more information about data and control flow may be used to reduce such conflicts.

2.2 Execute instructions

ESA/390 supports execute instructions, wherein a *subject instruction* at a specified address is read from memory, is modified by inserting fields from a register specified in the EX instruction, and then executed by the processor as part of the instruction stream.

A first pass implementation of the execute instruction capability consists of emitting code to

1. load the target instruction,
2. compose the instruction,
3. test for a particular bit pattern, and if successful,
4. execute translated code corresponding to that source operation, or,
5. if no match is found, invoke the translator to add a further test for the newly discovered bit pattern and its translation.

Several tests can be chained in a tree-shaped translation block, yielding an implementation similar to Mimic, where several EX-only blocks are compiled, and the respective assumption checks in several EX-only blocks are chained, until a valid translation is found.

While the described translation strategy offers a correct implementation for all possible cases of EX *subject instruction* evaluation, optimized code can be generated for a number of cases.

These cases correspond to a few idioms commonly found in ESA/390 code:

- EX is used to implement variable-length storage-to-storage operations by filling in the length field of a memory-to-memory (SS) operation subject instruction.
- EX is used to synthesize register-to-memory operations from immediate-to-memory (SI) operations by inserting the value stored in a general purpose register into the immediate field of an SI subject operation.

In these cases, the subject instruction is often included in the read-only constant pool of the function, and the combination of EX instruction and its subject SS or SI instruction can be translated directly into a variable length storage-to-storage operation, or a register-to-memory operation, respectively. Similar to the handling of self-modifying code, store operations into the EX subject instruction location would invalidate all translations which reference this location as EX subject.

2.3 Constant pool

The original System/370 architecture did not supply immediate-to-register operations. Instead, constants were stored in a per-function constant pool (similar to the Java Virtual Machine) and accessed with storage-to-register operations.

Immediate-to-register operations (RI) have only been recently introduced, and are not present in several important workloads. To optimize performance and facilitate optimization, references to read-only constant pools can be replaced by inline constants. However, to preserve 100% architectural compatibility (i.e., for setting reference bits and accurate page fault behavior), probing references need to be inserted into a translation group if data accesses are eliminated for data on pages other than the one holding the instruction stream for the current translation group(s).

3. HARDWARE SUPPORT

The DAISY VLIW processor core supports the execution of different instruction set architectures, such as PowerPC, ESA/390 and the Java Virtual Machine. Instruction prefetching provides the required instruction fetch bandwidth, and instrumentation support hardware is used to adapt to dynamically changing path profiles.

The DAISY long instruction word processor includes features such as control and data speculation, static out-of-order execution, and delayed exception handling. DAISY is based on a high-performance branch architecture which provides multiway branching capabilities in every cycle, to achieve good performance even on integer workloads with frequent control flow instructions.

The instruction format of the VLIW instructions is based on a limited variable length encoding, which was designed to reduce the complexity of the instruction alignment logic. In the encoding used for these experiments, VLIW instructions have a multiple of 4 operations and can either branch to the sequential VLIW instruction or perform a multiway branch.

Java data types and operations are very similar to those of PowerPC assuming a JIT compiler first compiles JAVA byte codes into RISC primitives and then parallelizes the primitives into VLIW instructions [2][10]. In the following sections we will focus on the issues involved in achieving commonality between S/390 and PowerPC.

The PowerPC 32, 16 and 8 bit integer data types are a superset of those of ESA/390. S/390 addresses can be 31 or 24 bits and their implementation will be discussed in subsection 3.3. A DAISY VLIW processor supporting multiple system binary translation will have support for both IBM and IEEE floating point formats, allowing it to implement both ESA/390 and PowerPC arithmetic operations. This does not introduce any major complications in the design of the floating point unit, since IEEE floating point subsumes the capabilities of IBM floating point formats. Execution primitives also include support for decimal excess-six arithmetic, which is important for some business-oriented ESA/390 applications.

3.1 Condition code flags

Most ESA/390 instructions set the condition code as the by-product of an arithmetic operation. Previous binary translation work has been mostly concerned with reducing the cost of materializing condition codes of one architecture on another architecture with dissimilar condition code semantics [9][11]. This is not a problem for a processor core specifically designed for binary translation of ESA/390, since condition codes can be architected to work with similar semantics.

A more serious problem associated with condition codes for this work is the introduction of an output and anti-dependency chain by the pervasive setting of the condition code. This serializes all instructions in a given ESA/390 program and limits the performance which can be obtained on a wide issue architecture through increased instruction level parallelism. To address this issue, the condition code computations must be eliminated, or the condition code must be renamed.

While most condition code results are never used, their computation is hard to eliminate in the context of 100% accurate architectural emulation because the accuracy of liveness analysis is limited in full emulation. Any instruction which can raise a synchronous exception represents a potential control flow to the exception handler. This includes all operations with memory operands, which can experience a page fault. As a result, traditional liveness analysis cannot eliminate many computations. In [12], we present a method for the deferred materialization of condition codes with zero execution time overhead.

Our deferred materialization approach is based on recording sufficient information to recreate the condition code value in the case of an exception being raised. However, unlike previous approaches, the recording is not done dynamically by copying parameters and the computing operation into a temporary space. Instead, the live ranges of the input parameters are extended to the end of the liveness range of the condition code computed from them. In addition, the operation used to compute the condition code value is recorded *at compile time* in a structure associated with the translation group.

These operations can be performed at compile time, so no execution time overhead is involved. However, since the information is static, this information cannot be carried across join points. Because in the DAISY translation approach, control flow joins occur only at the translation group boundaries (a translation unit is characterized by a single entry, multiple exits, and no join points inside the translation group), condition codes are fully materialized at group exits. Typically, groups are terminated by conditional branches, so condition code materialization at this point should not incur any additional cost.

Condition code renaming can occur using one of two approaches. In the results presented here, operation primitives to compute the condition codes according to the semantics of any given instruction are introduced. Each ESA/390 instruction is then “cracked” into multiple execution primitives which can be renamed and scheduled independently. To reduce code expansion, this can be combined with deferred materialization of condition codes based on static information [12]. Performance benefits which can be obtained from this deferred materialization are currently not modeled.

An alternative approach may rename condition codes in concert with ESA/390 general purpose registers, and compute the condition code in tandem with the integer result. The integer result and the condition code can then be committed to the processor state separately, or in a single operation. This treatment is similar to the treatment of the carry and overflow flags of the PowerPC XER register in the DAISY architecture [1].

3.2 Instruction atomicity

ESA/390 instructions are to be executed as atomic units, i.e., exceptions and interrupts are either recognized before or after the instruction appears to have executed. While atomicity is simple to achieve for RISC-style register-to-register operations as implemented by the convergence platform, instructions involving multiple memory operands pose several challenges. These are specifically related to the implementation of virtual memory, storage protection and asynchronous interrupts.

Atomicity rules are a particularly hard requirement for storage-to-storage (SS) operations, where all possible error conditions, e.g., page faults and protection faults in any part of the operands, must be detected before any architected state is altered.

To achieve the semblance of atomic behavior, DAISY/390 uses two approaches to deal with synchronous and asynchronous interrupts. Asynchronous interrupts are deferred until a later ESA/390 instruction boundary. Typically, these boundaries correspond to group boundaries, or other well defined points while the remaining code is executing with interrupts disabled. Since translation units are acyclic and of bounded height, there is an acceptable upper bound on the delay of such asynchronous interrupts corresponding to the maximum distance from a group entry to its exit.

Unlike asynchronous interrupts which can be delayed, synchronous exceptions need to be detected and serviced before any architected processor state, particularly memory, is altered. This is ensured by a combination of the binary translation platform and the translation strategy.

To support software-based instruction reordering and speculation, the DAISY register file contains registers corresponding to the architected machine state of the base architecture and additional rename registers. In addition, the DAISY processor supports silent and deferred exceptions which are propagated for speculative results [13], and ensures VLIW atomicity (i.e., either all or none of the results computed in a VLIW are committed to the processor state).

The translation strategy renames all re-ordered and/or speculatively computed results to a rename register, and commits them to the architected base architecture machine state in-order. Operations (such as memory stores) whose results cannot be renamed are executed in-order. Together with the atomicity of VLIWs, this ensures that processing can never skip *beyond* the precise exception point. When an exception is raised and the currently executing VLIW is nullified, an exception can be raised at a point corresponding to an instruction address preceding the precise exception point. Interpretation is then used from that point to find the actual exception point.

Memory-to-memory (SS) operations which work on memory ranges pose some additional requirements, since they generate multiple results which must be atomically committed. To ensure this behavior, additional code is inserted by the translator to probe all memory pages before actually changing any architected processor state. Consider the example of the MVC (move character) storage-to-storage operation which probes the end of the memory range (which could straddle a page boundary and raise an exception after processing the first page) before attempting to transfer the first byte:

```
MVC  L=256, 0(fp), 1024(fp)

⇒

LPROBE  255(fp)  // probe for reading
SPROBE  1279(fp) // probe for writing
for (i = 0...255) // copy loop
    LBZ   RTMP,i(fp)
    STB   RTMP,1024+i(fp)
```

The MVC instruction can move at most 256 bytes, while the ESA/390 page size is 4K, so two probe instructions to the ends of the source and destination storage operands, respectively, are enough to rule out further page faults during the execution of the MVC instruction. A special optimization is applicable to short memory-to-memory operations where all changes to the architected state can be performed in a single VLIW instruction. Since VLIW instructions exhibit the atomic execution property, no pre-probing of the operands is necessary. If a synchronous exception occurs, then the state remains unchanged and control can be passed to the exception handler. This execution property is important for the efficient translation of SS operations such as MVC (move), NC (and), OC (inclusive or), and XC (exclusive or) which operate on short memory data and which can be translated into very effective code without additional operand probing overhead. The binary translator must ensure that such operations are committed all together, which may entail skipping to the next VLIW if there is not enough space to commit them all in the current VLIW being filled.

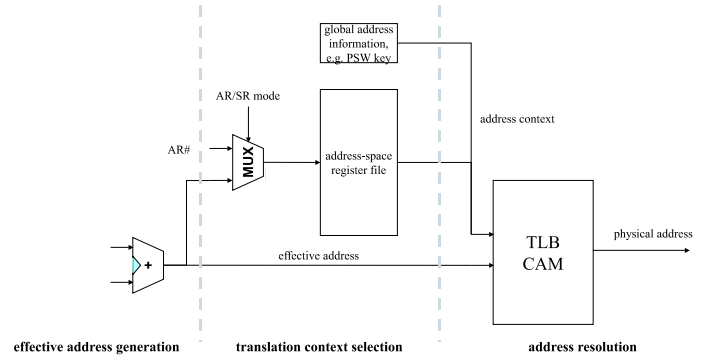


Figure 5: Common memory management architecture for emulating multiple base architecture addressing schemes.

3.3 Memory access semantics

An architectural challenge in the design of a processor supporting both PowerPC and ESA/390 execution is the correct and efficient implementation of the memory access model. These are quite different for the PowerPC and ESA/390 platforms.

The PowerPC architecture uses a segment-based addressing model. The segments are selected based on the high-order 4 bits of the effective address. In multiprocessor systems, PowerPC processors use a weak consistency model.

Due to its long history, ESA/390 supports multiple addressing schemes.

Addresses can be either 24 or 31 bits wide, and are interpreted according to several addressing modes. The most general of these modes uses access registers. Access registers are segment registers which are selected by the name of the base register used for address formation. (Thus if a memory address is computed using base register r12, the access register ar12 is used for resolving the effective to physical address mapping.) In multiprocessor implementations, ESA/390 is “firmly consistent”.³

To support both PowerPC and ESA/390 adequately, a migrant architecture must support both styles of address generation, memory addressing, and consistency. At its simplest, address formation consists of three steps:

effective address computation According to an architectures instruction set, an effective address is computed.

translation context selection The context used for a translation step is selected. The context can consist of several different aspects, such as whether address translation is enabled, the processor is in problem or system state, the segment referenced by a particular address, etc.

address resolution A CAM (content addressable memory) lookup is performed using the effective operation and the translation context. If the CAM lookup is successful, the physical address is returned. Otherwise, an exceptional condition is indicated.

We have designed a common memory management unit to address the issues posed by supporting segment register and access register based memory address translation by following the above analysis (see figure 5). The combined memory management unit implements the steps in a generic manner so as to map different base architectures onto this common MMU, such as PowerPC, ESA/390, or Intel x86.

³In firmly consistent memory, load operations on one processor may be moved above logically preceding store operations on the same processor. [14]

The address space identifier register file can be accessed with either the high order effective address bits to support effective-address based segment selection (e.g., as used in PowerPC), or by using an explicit segment specifier inside the VLIW load/store primitive (e.g., using the access register number in ESA/390). The SR/AR selector can be specified either using two distinct flavors of load operations, or a global mode bit in the processor state.

The address space identifier file does not necessarily hold the same values as segment or access registers hold. Their only use is in providing address bits which are used to match in the TLB CAM, i.e., there would typically be a mapping between each address space and the contents of an address specifier register.

The TLB CAM used to perform the actual translation step can consist of a multi-level cache-based structure to combine low latency operation with high hit rates. The TLB CAM is reloaded by a software-based reload mechanism to efficiently support multiple base architectures. The software-based page table reload handler is invoked on a TLB CAM miss and interprets the page tables following the architecture specifications of the base architecture.

Strong memory consistency (sequential consistency) can be achieved using the load data verification scheme described in [15]. This scheme performs a load and verify operation at the in-order point for any load operation which has been performed out-of-order with respect to other memory operations. If a data difference is found, fixup code is invoked to re-execute the load operations and all dependent operations. Strong memory consistency subsumes ESA/390's firm consistency and the weaker PowerPC memory semantics.

3.3.1 PowerPC compatible operation

In PowerPC-compatible operation, the effective address is generated by adding displacement and base registers. The address space identifier file is selected using the high order 4 bits of the effective address. In addition, global context information selects different mappings according to whether address translation has been enabled or not.

If a TLB CAM miss occurs, a native VLIW exception occurs and the software page fault handler interprets the page tables according to the PowerPC page table format.

3.3.2 ESA/390 compatible operation

In ESA/390-compatible operation, the address generation step can include the application of a 31 or 24 bit address mask to reduce the address width in accordance to the selected ESA/390 addressing mode. This effective address is then translated using address-space specific page tables according to several different address translation modes.⁴ The page tables are indicated as translation context and the TLB CAM only contains fully resolved effective to physical address mappings.

The access register number is used to access the address space identifier file which provides additional context information. The global address context supplies further bits, such as the PSW storage key which is used to implement the ESA/390 storage protection model.

The TLB CAM then tries to translate a triple consisting of effective address, address space specifier and global context to a physical address. A miss in the translation can occur for a number of reasons. A miss can indicate that the software page table reload function needs to be invoked because no translation for the given effective address has been loaded for the specified address space. Alternatively, a mismatch in the PSW storage key section may have caused a tag

⁴For the sake of simplicity, we will explain operation only of the most complex, access register based mode. All other memory translation modes can be reduced to the access register mode.

mismatch, and the appropriate reaction is then to invoke the base ESA/390 storage key protection mechanism.

Several performance improvements are possible to increase the performance of the merged MMU (at the cost of generality). These include adding a special mode bit to ignore a number of high order bits in the tag match of the TLB CAM, thereby eliminating the need for address truncation in the effective address generation phase. (Alternatively, multiple mappings could be maintained in the TLB CAM which represent the same 24 bit address with different 8 bit prefixes mapping to the same physical address.)

3.3.3 Compatibility with other architectures

The common MMU facility can also be used to translate other memory translation models, such as Intel x86. In Intel x86 mode, the access address specifier register would be selected based on the instruction-specific segment (or the segment override prefix). Similarly, other addressing modes can be reduced to the present MMU design.

4. PERFORMANCE OF A BINARY TRANSLATION PLATFORM

Binary translation performance is composed of several aspects, such as the performance of the target architecture and the cost of performing the translation. Combining these different performance aspects provides an overall view of system performance. This approach is similar to the MACS approach of characterizing different aspects of system performance [16]. Unlike the MACS approach, these bounds are derived not by experimental means, but by simulation and analytical modeling of a binary translation system.

To model the different aspects of binary translation performance, we have developed a trace-based tool chain coupled with some analytical modeling to explore the performance potential. Trace-based analysis allows the evaluation of arbitrary program sequences, including operating system code.

The results presented here were collected under the OpenMVS operating system, which provides a hosted POSIX execution environment under MVS. Results include not only the application code, but also operating system code. Thus, this approach gives access to a more realistic workload model compared to user-level application code which is often used in such experiments.

4.1 Components of Performance

A first cracking step decomposes the CISC instruction stream into RISC-like primitive operations, which are then used by all further processing steps. Using simple instruction primitives offers increased flexibility in subsequent scheduling and optimization steps.

The number of primitive operations generated for each source instruction is a measure of both the architectural fit between the base and the target platform, as well as the architectural complexity of the base architecture. The expansion ratio from complex ESA/390 instructions to VLIW primitives is shown in 2.

When translation groups (which take the form of tree regions) are first formed, a measure we refer to as infinite resource CPI is used to guide the group formation. The infinite resource CPI within a particular path of a group, is equal to the critical dependence chain length in this path divided by the number of ESA/390 instructions in the path. Such CPI could be theoretically obtained by an infinite resource machine, and serves as a CPI lower bound. By building larger translation groups, more scheduling opportunities are discovered, resulting in a lower overall infinite resource CPI, which in turn can (if instruction cache penalties are managed) lead to lower overall finite resource CPI.

By scheduling the translation groups for a specific target machine, the constrained machine CPI is obtained. This accounts for the lim-

its imposed by the machine architecture, such as the number of ALUs, FPUs, load/store units, and communication costs between clusters of ALUs and register files. This adder also includes instructions added for system operation, such as commit operations to the machine state, multiprocessor consistency, and translation consistency (which must be checked when crossing instruction pages).

A finite instruction cache adder accounts for pipeline stall cycles due to instruction cache misses. Instruction cache performance is impacted by the translation strategy and the optimization performed. Instruction cache performance is directly proportional to reuse and indirectly proportional to code size. Code size is a function of VLIW encoding and of code duplication. Thus, while more sophisticated translation schemes generate longer groups with higher infinite cache ILP, this is counterbalanced by instruction cache penalties due to code duplication and lower re-use.

A number of hardware design decisions have been made particularly to address this issue. To reduce the impact of code size, we use a variable length instruction encoding. In addition, instruction cache prefetching for fetching the two most recently used successor lines is employed to reduce the miss rates.

Instruction cache performance is further impacted by the operation of the binary translator. Every time the translator is invoked, it will displace the translated code from the instruction cache. We model this by computing the rate at which the translator is invoked, and periodically flushing the first-level instruction cache.

The average rate of translation events is computed as follows

$$\text{translation interval in VLIWs} = \frac{\text{reuse rate} * \text{primitives}}{\text{group formation events}} * \text{primitive CPI}_{\text{infinite cache}}$$

The finite data cache adder (and finite data TLB adder) accounts for pipeline stalls due to data cache misses (and data TLB misses). Compared to in-order execution of memory operations, speculation and memory reordering add burden to the memory hierarchy. For our current coherence and disambiguation approach [15], experiments show an overhead of about 70%. We account for this overhead with a speculation factor 1.7, i.e., we multiply by 1.7 the number of stall cycles reported by the (non-speculating) memory simulator. Since the target architecture implements a stall-on-miss policy, modeling data cache effects is purely additive.

Like the instruction cache, the data cache is impacted by the operation of the translator. The translator accesses a number of data structures through the data cache thereby displacing the data of the base system. Data accessed by the binary translator includes: ESA/390 code, intermediate code representation, generated native VLIW code, as well as meta data such as group descriptors mapping groups to starting addresses and mapping memory areas to groups.

The average rate of translation events is computed as follows

$$\text{translation interval in data refs} = \frac{\text{reuse rate} * \text{memory operations}}{\text{group formation events}}$$

The actual cost of translating instructions is computed using an analytical model based on the number of translations which have occurred. When primitives are first included in a group, each translation of a primitive is assumed to take 4000 cycles to account for decoding, optimization, etc. This is a conservative estimate based on our experience with DAISY which yielded about 4000 PowerPC operations to translate one *PowerPC* operation. When a group is later extended, these instructions have already been translated, so they only need to be decoded and rescheduled at a lower cost of 800 cycles. Our accounting of the cost per primitives reflects the higher cost of translating more complex instructions which expand into multiple primitives.

The CPI contribution of the translator is then

$$CPI_{TR} = \frac{\text{primary translations} * 4000 + \text{secondary translations} * 800}{\text{reuse rate}}$$

A final CPI degradation is due to exception overhead. To reduce the number of groups starting at arbitrary instruction addresses (i.e., at various instruction addresses where an exception was received), interpretive execution is entered after executing a return from the exception handler until an existing group is encountered. This overhead is modeled as follows:

$$CPI_{EXC} = \frac{\text{interpretation cost} * \text{unique primitives}}{\text{exception rate} * \text{total groups} * 2}$$

4.2 Evaluation Tool Chain

System performance of a binary translation architecture is evaluated using a tool chain of several modeling tools which model the various performance aspects of a dynamic compilation system. The data generated by these tools are combined with analytical models to derive overall system performance.

The system evaluation tool chain consists of the following modeling tools:

group former A tree-region former cracks complex ESA/390 instructions into instruction primitives and forms tree-regions according to the execution-based strategy described previously in [7].

The tree-region former models the initial group formation and group extension and computes infinite resource CPI information which is used to guide the group formation process.

VLIW scheduling A VLIW scheduler schedules the VLIW primitives in each tree region and generates VLIW code according to the clustering, functional unit and register constraints. Tree-region based optimizations as well as speculation is performed during this phase.

The VLIW scheduler also determines the number of cycles for each exit (tip) of a tree-region. When the entry to exit cycle count for the various paths through a tree region is combined with program control flow information gathered by the tree-region former (tip trace), a finite resource CPI can be computed.

VLIW memory layout A VLIW instruction memory layout tool lays out VLIWs in memory according to architecture requirements.

I-cache simulation A multi-level instruction cache simulator performs instruction stream prefetching and *hit/miss* simulation and computes the CPI contribution of instruction memory access.

D-cache/D-TLB simulation A multi-level data cache and data TLB simulator performs hit/miss simulation and computes the CPI contribution of data memory accesses.

ILP measurement has to be adjusted for the difference in complexity between typical RISC and CISC architectures. This adjustment cannot be achieved by merely lowering the ILP goal. Due to the large variance in complexity in CISC instructions, it is important to account for the non-uniform distribution of instruction complexity. This can lead to high peak of instruction issue demands to achieve the requested ILP.

Cache	Size	Linesize	Assoc	Latency
L1-I	32K	1K	8	1
L2-I	1.5M	2K	6	3
L1-D	32K	256	4	3
L2-D	512K	256	8	6
L3	∞			
DTLB1	128 Entries	–	2	2
DTLB2	1K Entries	–	8	4
DTLB3	∞	–		

Table 1: Cache and TLB Parameters.

Without some smoothing, even a moderate ILP goal may lead to massive code explosion as groups containing a high complexity CISC instruction are extended to compensate for the low S/390 CPI of complex instructions. This can lead to massive code explosion without any significant gains in instruction level parallelism.

An example of such high-complexity regions are string and memory block operations. On typical RISC architectures, such operations are represented as loops consisting of primitive memory operations. On ESA/390, these operations are often coded to use a single CISC memory-to-memory operation such as MVC (move character string), CLC (compare character string), and TRT (translate and test, used to locate a byte in a string by string operations such as strlen or index). An MVC instruction may require 512 RISC primitives. Thus to attain ESA/390 ILP of 3, a machine which could issue 1536 primitives at once would be required.

We have addressed this problem by internally measuring ILP in primitives as generated by the instruction cracking step. The primitive goal is derived from the ESA/390 ISA ILP goal by multiplying the ESA/390 goal with the dynamic instruction expansion factor. This dynamic expansion factor is derived as

$$\text{expansion factor} = \frac{\text{primitives per trace segment}}{\text{ESA/390 instructions per trace segment}}$$

The expansion factor can be determined either dynamically, or as a fixed constant based on typical workload characteristics.

5. RESULTS AND DISCUSSION

To establish the performance potential of the described approach, we modeled several integer workloads from the SPECint95 benchmark suite and an interactive workload containing significant portions of system level code (device drivers, network, I/O) in addition to a user workload. Binary translation was targeted at a DAISY VLIW architecture executing a tree-VLIW with up to 16 operations and a multiway branch per cycle.

Cache and TLB parameters are given in table 1. To achieve high throughput and low latency for instruction fetch, DAISY uses a partitioned instruction cache consisting of separate “mini I-caches” for each cluster of two ALUs, which contain the instructions for that cluster as well as some duplicated control information. This reduces the wire length during cache access and allows to build faster memory arrays and large logical line sizes. In addition, each VLIW encodes the next line address to further reduce fetch latency.

Experiments were devised to specifically explore different trade-offs in the system operation, such as group formation strategies or the impact of ESA/390 specific optimization opportunities. A first set of experiments compares two different levels of eagerness in group formation. Both levels use the same settings for initial code creation, based on a translation path length of 24 primitives and an initial infinite resource primitive ILP goal of 9.

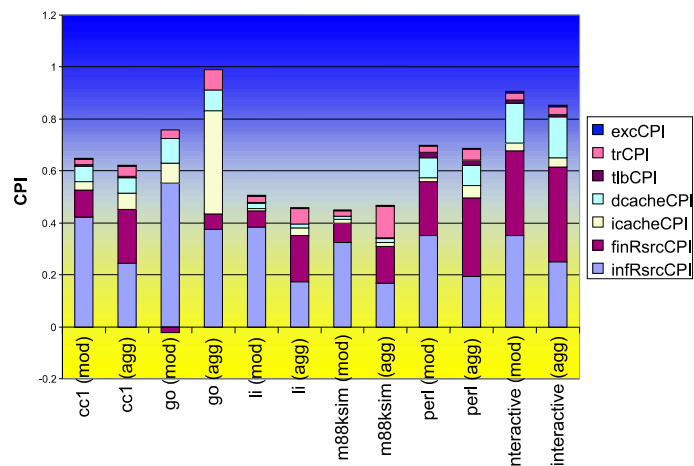


Figure 6: Group extension aggressiveness: more aggressive extension of groups yields higher infinite resource ILP. This is counterbalanced by finite resource constraints and instruction cache effects. To achieve optimal performance, a more aggressive hardware design and good control over instruction cache effects are necessary.

In the moderate group extension approach, a code segment has to account for 5% of the execution to be eligible for group extension. The path length limit on during group enlargement is 180 primitives. A more aggressive group extension strategy extends code segments when they account for 0.1% of the total execution, with a 250 primitive limit on path length.

Figure 6 compares the two extension approaches. Aggressive group formation shows significant improvement in infinite resource CPI. (This comes at the expense of somewhat increased translation time since more group formation events occur.) Infinite resource ILP appears to scale logarithmically with path length, so significant increases in pathlength are necessary to improve performance noticeably.

However, much of the improvement in infinite ILP is lost when the instructions are scheduled for a 16-way VLIW machine, making the case for an even wider, more aggressive hardware platform. However, as previously mentioned, another aspect of binary translation performance is code expansion control. In all cases, instruction cache performance is worse for aggressive group formation.

This effect is most pronounced for the go benchmark. Since go has very unpredictable branching behavior, group formation builds groups with low reuse. Aggressive group extension further exacerbates this problem leading to even higher code duplication and lower reuse numbers. Many early group exits (as demonstrated by the low average dynamic path length spent in a group), unpredictable cache prefetching, and low cache re-use are the result. Thus, while other benchmarks compensate for higher code expansion by improved infinite cache ILP due to improved average dynamic group length, go suffers from code expansion without the benefits derived by other benchmarks.

These results indicate that binary translation performance is very susceptible to the workload. Thus, an adaptive response to measured system performance may be desirable. By using adaptive feedback from measured system behavior, excessive instruction cache miss rates and/or early group exits may call for a reduction in group formation aggressiveness while high prediction rates and low instruction cache misses may indicate an opportunity to extend groups and increase optimization increases.

We have computed the overhead translation and exception handling

CPI components directly due to the binary translation scheme with worst case parameters of a low reuse rate (10^6) and high synchronous exception rate (one exception every $20 * 10^3$ instructions). While these parameters are more conservative than actually observed program behavior, the CPI degradation due these adders is still low.

Experiments further indicate that the conversion of register-indirect branches to relative branches based on incremental dataflow analysis improved performance up to 15% compared to a baseline using the previously described optimizations for register-indirect branches [7].

Table 2 summarizes trace statistics and performance results for the selected benchmarks. The table gives the code expansion in processor pages referenced. The expansion is affected by a number of parameters, which include the instruction encoding, the number of VLIW primitives generated per CISC instruction, and the code duplication. This is counterbalanced by improvement of code density which is due to the fact that only those portions of a page which are actually executed are translated and stored in the VLIW translation cache.

We also report average dynamic pathlength, i.e., the number of cycles which execution actually spends in a typical code fragment. Dynamic average pathlength is a function of the group size and the branch predictability. As branch predictability erodes across successive branches, the likelihood of taking a path which is not included in the group grows. Average dynamic group length correlates highly to overall performance since optimizations, speculation and instruction scheduling are performed at the group level.

The primitive expansion column lists the average number of VLIW primitives generated for each CISC instruction.

6. RELATED WORK

Previous work in inter-system binary translation has largely focused on easing migration between platforms. As a result, performance goals are typically more moderate, i.e., to achieve acceptable performance levels, not to exceed native implementations.

Several systems attempt to emulate various portions of the Windows on Intel x86 system to execute binaries on alternative platforms. The target platforms are typically RISC platforms which were never designed for either binary translation or ease of x86 compatibility. In fact, there is often a significant mismatch between the Intel x86 and the target platform. Typical problems consist of floating point formats (accurate emulation of x86 code requires an 80 bit floating point format), the provision of appropriate primitives for computing the condition flags according to x86 semantics, etc.

Several techniques have been developed for coping with these issues, such as determining when it is safe to use native condition code computation or the (dynamically) deferred materialization of condition codes wherein condition code setting operations and its operands are recorded in registers and only materialize if they are used at a later point (a form of un-speculation [17]). In the case of DAISY, we have resolved these issues by defining appropriate primitives where a software solution seemed expensive. This is in keeping with the goal of achieving *better* performance through binary translation than through native implementation.

Another difference from previous approaches is how we deal with system issues. Previous systems typically operate in user mode and use a host operating system to provide services such as paging and device access. In contrast, a system executing under DAISY natively manages such devices, requiring exact access behavior for memory-mapped I/O accesses, reporting of page faults, etc.

Several approaches exist to providing system services to programs running under binary translation systems:

service identity The DEC/Compaq FX!32 x86 to Alpha translator [18] relies on the identity between OS services offered by Microsoft Windows NT on different platforms, and dispatches each function to a corresponding function on the native operating system.

ABI interface emulation Sun's WABI [11] intercepts calls to the Windows ABI and implements similar functionality using native libraries.

execution of native OS Systems such as SoftWindows or SimOS [19] boot the operating system and use virtual devices to access the services of the host operating system.

Before the inception of the DAISY project, no machines have been designed exclusively as target platforms for binary translation. The DEC/Compaq Alpha was however designed to ease migration from the VAX architecture, and offered a number of compatibility features. These include similar memory management capabilities to ease migration of the VAX/VMS operating system, and support for VAX floating point formats. DEC's original transition strategy called for static binary translators to support program migration. Two translators supported these migration strategy: VEST for VAX/VMS migration to Alpha/OpenVMS and mx for migration from DEC Ultrix on the MIPS architecture to OSF1 on DEC Alpha [20]. Later, the FX!32 dynamic binary translator was added to ease migration from Windows on x86 to Windows on Alpha.

Recently, Transmeta has announced an implementation of the Intel x86 processor based on binary translation to a VLIW processor [21]. The processor described is based on a VLIW with hardware support for checkpointing architected processor state to implement precise exceptions using a rollback/commit strategy. Rollback of memory operations is supported using a gated store buffer [22].

Several hardware implementation schemes are related to the outlined software scheme. The Pentium Pro and AMD K6 perform translation to RISC-like execution primitives. This allows the exposure of parallelism within CISC instructions similar to the approach taken in this work. However, the amount of re-arranging of code which can be performed to achieve high instruction level parallelism is limited due to hardware complexity.

The present approach is different from the DIF approach of Nair and Hopkins [23]. Our approach schedules operations on multiple paths to avoid serializing due to mispredicted branches. Also, in the present approach, there is virtually no limit to the length of a path within a tree region or the ILP achieved. In DIF, the length of a (single-path) region is limited by machine design constraints (e.g., 4-8 VLIWs). Our approach follows an all software approach as opposed to DIF which uses a hardware translator. This all-software technique allows aggressive software optimizations hard to do by hardware alone. Also, the DIF approach involves almost three machines: the sequential engine, the translator, and the VLIW engine. In our approach there is only a relatively simple VLIW machine.

Trace processors [24] are similar to DIF except that the machine is out-of-order as opposed to a VLIW. This has the advantage that different trace fragments do not need to serialize between transitions between one trace cache entry and another. However, when the program takes a path other than what was recorded in the trace cache, a serialization can occur. The present approach solves this problem by incorporating an arbitrary number of paths in a software trace cache entry, and by very efficient zero overhead multiway branching hardware [25].

7. CONCLUSION

Benchmark	Group Fm'n	trace statistics			CPI components							inf. cache CPI	fin. cache CPI
		Code Exp.	avg. Pthlen	prim. exp.	inf Rsrc	fin Rsrc	insn cache	data cache	TLB	trans- lation	exc		
cc1	agg	1.78	105	1.62	0.25	0.21	0.06	0.06	0.01	0.04	0.00	0.45	0.62
cc1	mod	1.78	44	1.62	0.42	0.10	0.03	0.06	0.01	0.02	0.00	0.53	0.65
go	agg	44.95	55	1.36	0.38	0.06	0.39	0.08	0.00	0.08	0.00	0.44	1.00
go	mod	14.72	30	1.36	0.55	-0.02	0.08	0.08	0.00	0.03	0.00	0.53	0.73
li	agg	3.97	214	1.59	0.18	0.18	0.03	0.02	0.00	0.07	0.00	0.35	0.48
li	mod	1.14	46	1.59	0.39	0.06	0.01	0.02	0.00	0.02	0.00	0.45	0.50
m88ksim	agg	9.48	202	1.52	0.17	0.14	0.02	0.01	0.00	0.12	0.00	0.31	0.46
m88ksim	mod	1.60	55	1.52	0.33	0.07	0.01	0.02	0.00	0.02	0.00	0.40	0.46
perl	agg	2.70	131	1.81	0.19	0.30	0.05	0.08	0.02	0.04	0.00	0.49	0.68
perl	mod	1.29	54	1.81	0.35	0.21	0.01	0.08	0.02	0.02	0.00	0.55	0.69
interactive	agg	1.25	127	2.49	0.25	0.36	0.03	0.15	0.01	0.03	0.01	0.61	0.84
interactive	mod	1.00	79	2.49	0.35	0.32	0.03	0.16	0.01	0.03	0.01	0.67	0.89

Table 2: Preliminary CPI results are reported as cycles per ESA/390 instructions.

In this paper we have attempted to address in detail the issues in implementing a binary translation VLIW for a very complex legacy architecture. S/390 introduces many unusual problems that need to be solved while building an aggressive wide issue processor, that are not common in newer RISC architectures. These problems include self modifying code, atomicity of complex CISC instructions in the presence of precise interrupts, reordering memory operations in the presence of MP memory consistency, access registers, pervasive use of indirect branches, execute instructions, single condition code set by almost all operations, program event recording, and so on.

Our solutions in this paper have included the use of incremental dataflow information to determine the target addresses of register indirect branches and maintaining precise exceptions at CISC instruction boundaries. We achieve this by pretesting for exception situations before modifying architected state. Additional optimization possibilities are the elimination of spurious condition code computations using a compilation scheme based on deferred materialization. A VLIW exception handler materializes “dead” condition codes which are visible to exception handlers only before transferring control to a translation of the native exception handler. Attacking this package of unusual points allows us to deal with processor design issues which are quite important in practice, although not typically considered by research studies on novel processor architectures.

Next, we have taken steps toward designing common VLIW hardware to support different architectures such as PowerPC, S/390 and the Java Virtual machine. Such a convergence architecture has the potential to become a new kind of open system, where “computer architectures” are in fact software layers on a single generic, simple wide issue engine, which can lead to significant cost savings and flexibility. While previous work did state goals of hardware commonality and convergence [1][3], this is the first paper to address binary-translation based hardware convergence issues of multiple architectures in detail.

In addition to primitives supporting the multiple layered architectures, our design includes a configurable memory management unit which can be used to emulate different memory addressing semantics, such as segmentation and access register based models. This approach is based on a software-managed TLB which translates effective to physical addresses under the control of architecture-specific translation context bits which are managed in software. The software TLB handlers are responsible for appropriately selecting the address translation context and interpreting page tables.

Third, we have performed a detailed trace based preliminary study for measuring the performance potential of such a convergence architecture, including cache, TLB, translation, interpretation and ex-

ception CPI overheads. While further study is required, the initial results are encouraging. The performance potential is based the ability to decompose complex CISC operations into simpler operations primitives which can be scheduled in parallel. Strategies used to boost performance are profile-directed feedback into the group formation process to adapt to changing program behavior, collection of dataflow information during the binary translation process, and ILP compilation techniques to increase available instruction level parallelism.

8. ACKNOWLEDGMENTS

The authors wish to thank Dave Luick for many valuable suggestions.

9. REFERENCES

- [1] K. Ebcioglu and E. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. Research Report RC 20538, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1996.
- [2] K. Ebcioglu, E. R. Altman, and E. Hokenek. A JAVA ILP Machine based on Fast Dynamic Compilation. In *IEEE MASCOTS International Workshop on Security and Efficiency Aspects of Java*, January 1997.
- [3] J. E. Smith, T. Heil, S. Sastry, and T. M. Bezenek. Achieving high performance via co-designed virtual machines. In *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 77–84, October 1998.
- [4] G. M. Silberman and K. Ebcioglu. An Architectural Framework for Migration from CISC to Higher Performance Platforms. In *Proc of the 1992 International Conference on Supercomputing*, pages 198–215, Washington, DC, July 1992. ACM Press.
- [5] G. M. Silberman and K. Ebcioglu. An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures. *IEEE Computer*, 26(6):39–56, June 1993.
- [6] K. Ebcioglu and E. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 26–37, Denver, CO, June 1997. ACM.

- [7] K. Ebcioğlu, E. Altman, S. Sathaye, and M. Gschwind. Execution-based scheduling for VLIW architectures. In *Euro-Par '99 Parallel Processing – 5th International Euro-Par Conference*, number 1685 in Lecture Notes in Computer Science, pages 1269–1280. Springer Verlag, Berlin, Germany, August 1999.
- [8] K. Ebcioğlu, E. Altman, S. Sathaye, and M. Gschwind. Optimizations and oracle parallelism with dynamic translation. In *Accepted for: Micro-32*, Haifa, Israel, November 1999.
- [9] C. May. Mimic: A fast S/370 simulator. In *Proc. of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques*, volume 22 of *SIGPLAN Notices*, pages 1–13. ACM, June 1987.
- [10] S. Kim, S.-M. Moon, K. Ebcioğlu, and E. Altman. VLaTTe: a Java just-in-time compiler for VLIW with fast scheduling and register allocation. In *preparation*.
- [11] P. Hohensee, M. Myszewski, and D. Reese. WABI CPU emulation. In *Hot Chips VIII*, Palo Alto, CA, 1996.
- [12] M. Gschwind. Method for the deferred materialization of condition code information. *Research Disclosures*, 1999. (to appear).
- [13] K. Ebcioğlu. Some Design Ideas for a VLIW Architecture for Sequential-Natured Software. In M. Cosnard et al., editor, *Parallel Processing*, pages 3–21. North-Holland, 1988. (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing).
- [14] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, December 1996.
- [15] J. Moreno and M. Moudgill. Method and apparatus for re-ordering of memory operations in a processor. US Patent No. 5,758,051, May 1998.
- [16] E. Boyd and E. Davidson. Hierarchical performance modeling with MACS: a case study of the Convex C-240. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, pages 203–210, San Diego, CA, May 1993. ACM.
- [17] K. Ebcioğlu, R. Groves, K. Kim, and G. Silberman. VLIW compilation techniques in a superscalar environment. In *Proc. of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, volume 29 of *SIGPLAN Notices*, pages 36–48, Orlando, FL, June 1994. ACM.
- [18] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. FX132—A Profile-Directed Binary Translator. *IEEE Micro*, 18(2):56–64, March 1998.
- [19] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, 3(4):34–43, Winter 1995.
- [20] R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.
- [21] A. Klaiber. The technology behind crusoe processors. Technical report, Transmeta Corp., Santa Clara, CA, January 2000.
- [22] E. Kelly, R. Cmelik, and M. Wing. Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed. US Patent 5832205, November 1998.
- [23] R. Nair and M. Hopkins. Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups. In *Proc of the 24th Annual International Symposium on Computer Architecture*, pages 13–25, Denver, CO, June 1997. ACM.
- [24] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace Processors. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, Research Triangle Park, NC, December 1997. IEEE Computer Society.
- [25] K. Ebcioğlu, J. Fritts, S. Kosonocky, M. Gschwind, E. Altman, K. Kailas, and T. Bright. An eight-issue tree-VLIW processor for dynamic binary translation. In *Proc. of the 1998 International Conference on Computer Design (ICCD '98) – VLSI in Computers and Processors*, pages 488–495, Austin, TX, October 1998. IEEE Computer Society.